2

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

AD-A243 377

age 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data ling this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

REPORT DATE	3. REPORT TYPE AND DATES COVERED Final: 12 Dec 1990 tp 01 Jun 1993
4. TITLE AND SUBTITLE Alsys, AlsyCOMP_043, Version 5.3, Macintosh IIcx (Host & Target), 901221W1.11104	5. FUNDING NUMBERS
6. AUTHOR(S) Wright-Patterson AFB, Dayton, OH USA	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Ada Validation Facility, Language Control Facility ASD/SCEL Bldg. 676, Rm 135 Wright-Patterson AFB, Dayton, OH 45433	8. PERFORMING ORGANIZATION REPORT NUMBER AVF-VSR-441.0891
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office United States Department of Defense Pentagon, Rm 3E114 Washington, D.C. 20301-3081	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Alsys, AlsyCOMP_043, Version 5.3, Wright-Patterson AFB, Macintosh IIcx (Host & Target),ACVC 1.11.

91-16068

14. SUBJECT TERMS Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.	15 NUMBER OF PAGES 16 PRICE CODE

17 SECURITY CLASSIFICATION OF REPORT	18 SECURITY CLASSIFICATION	19 SECURITY CLASSIFICATION OF ABSTRACT	20 LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

NSN 7540-01-280-550

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-128

91 [illegible]

AVF Control Number: AVF-VSR-441.0891
1 August 1991
90-10-24-ALS

Ada COMPILER VALIDATION SUMMARY REPORT: Certificate Number: 901221W1.11104 Alsys AlsyCOMP_043, Version 5.3 Macintosh IIcx => Macintosh IIcx

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 12 December 1990.

Compiler Name and Version:	AlsyCOMP_043, Version 5.3
Host Computer System:	Macintosh IIcx, Macintosh System Software 6.0.5
Target Computer System:	Macintosh IIcx, Macintosh System Software 6.0.5
Customer Agreement Number:	90-10-24-ALS

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901221W1.11104 is awarded to Alsys. This certificate expires on 1 March 1993.

This report has been reviewed and is approved.

[illegible]

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

[illegible]

[illegible] Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

[illegible]

[illegible] Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

CUSTOMER:	Alsys, Inc.
ADA VALIDATION FACILITY:	Ada Validation Facility (ASD/SCEL) Computer Operations Division Information Systems and Technology Center Wright-Patterson AFB OH 45433-6503
ACVC VERSION:	1.11
ADA IMPLEMENTATION:	
COMPILER NAME AND VERSION:	AlsyCOMP_043 Version 5.3
HOST COMPUTER SYSTEM:	Apple Macintosh IIcx Macintosh System Software 6.0.5
TARGET COMPUTER SYSTEM:	Apple Macintosh IIcx Macintosh System Software 6.0.5

CUSTOMER'S DECLARATION:

I, the undersigned, representing Alsys, Inc., declare that Alsys, Inc. has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration.

[illegible] 90-12-20

Mike Blanchette,
Vice President, Engineering
Alsys, Inc.
67 South Bedford Street
Burlington, MA 01803-5152

Date

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 11 November 1990.

E28005C	B28006C	C34006D	C35702A	B41308B	C43004A
C45114A	C45346A	C45612B	C45651A	C46022A	B49008A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
B83026B	B85001L	C83026A	C83041A	C97116A	C98003B
BA2011A	CB7001A	CB7001B	CB7004A	CC1223A	BC1226A
CC1226B	BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A
CD2A21E	CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A
CD2B15C	BD3006A	BD4008A	CD4022A	CD4022D	CD4024B
CD4024C	CD4024D	CD4031A	CD4051D	CD5111A	CD7004C
ED7005D	CD7005E	AD7006A	CD7006E	AD7201A	AD7201E
CD7204B	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

The following 21 tests check for the predefined type LONG_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45536A, C46013B, C46031B, C46033B, and C46034B contain 'SMALL representation clauses which are not powers of two or ten.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten TYPE'SMALL; this implementation does not support decimal 'SMALLs. (See section 2.3.)

C45624A checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 5. For this implementation, MACHINE_OVERFLOWS is TRUE.

C45624B checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 6. For this implementation, MACHINE_OVERFLOWS is TRUE.

D55A03G..H (2 tests) use 63 levels of loop nesting which exceeds the capacity of the compiler.

C86001F recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

B86001Y checks for a predefined fixed-point type other than DURATION.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A840 use representation clauses specifying non-default sizes for access types.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

EE2401D, EE2401G, and CE2401H use instantiations of DIRECT_IO with unconstrained array and record types; this implementation raises USE_ERROR on the attempt to create a file of such types.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

The tests listed in the following table are not applicable because the given file operations are not supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2105A	CREATE	IN_FILE	SEQUENTIAL_IO
CE2105B	CREATE	IN_FILE	DIRECT_IO
CE3109A	CREATE	IN_FILE	TEXT_IO

The following 7 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file and more than one is open for writing; USE_ERROR is raised when this association is attempted.

CE2107B	CE2107D..E	CE2107G	CE2107L	CE3111D..E

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

CE2401H raises USE_ERROR when CREATE with mode INOUT_FILE is used for unconstrained records with default discriminants.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 18 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B23004A	B24007A	B24009A	B28003A	B32202A	B37004A
B61012A	B95069A	B95069B	B97103E	BA1101B	BC2001D
BC3009A	BC3009C				

B85002A was graded passed by Evaluation Modification as directed by the AVO. This test declares a record type REC2 whose sole component is of an unconstrained record type with a size in excess of 2**32 bytes; this implementation rejects the declaration of REC2. Although a strict interpretation of the LRM requires that this type declaration be accepted (an exception may be raised on the elaboration of the type or an object declaration), the AVO accepted this behavior in consideration that such early error detection is expected to be allowed by the revised language standard.

BA2001E was graded passed by Evaluation Modification as directed by the AVO. The test expects that duplicate names of subunits with a common ancestor will be detected as compilation errors; this implementation detects the errors at link time, and the AVO ruled that this behavior is acceptable.

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-10 value as 'SMALL for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal 'SMALLs may be omitted.

EA3004D was graded passed by Evaluation and Processing Modification as directed by the AVO. The test requires that either pragma INLINE is obeyed for a function call in each of three contexts and that thus three library units are made obsolete by the re-compilation of the inlined function's body, or else the pragma is ignored completely. This implementation obeys the pragma except when the call is within the package specification. When the test's files are processed in the given order, only two units are made obsolete; thus, the expected error at line 27 of file EA3004D6M is not valid and is not flagged. To confirm that indeed the pragma is not obeyed in this one case, the test was also processed with the files re-ordered so that the re-compilation follows only the package declaration (and thus the other library units will not be made obsolete, as they are compiled later); a "NOT APPLICABLE" result was produced, as expected. The revised order of files was 0-1-4-5-2-3-6.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

Mike Blanchette
67 South Bedford Street
Burlington MA 01803-5152

For a point of contact for sales information about this Ada implementation system, see:

Jerry Rudisin
67 South Bedford Street
Burlington MA 01803-5152

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a) Total Number of Applicable Tests	3793	
b) Total Number of Withdrawn Tests	83	
c) Processed Inapplicable Tests	93	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	201	
f) Total Number of Inapplicable Tests	294	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 294 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded onto a VAX 3100 and transferred to the host by FTP.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

Option	New value	Default
CALLS	INLINED	NORMAL

This option directs the compiler to obey the advice of INLINE pragmas.

ERRORS	999	50

This option directs the compiler to quit only after 999 source errors.

MEMORY	2000	300

This option directs the compiler to use more memory for internal buffering, reducing compilation time.

SHOW	NONE	ALL

This option directs the compiler to not produce banner information in the listing file. This is done to facilitate listing comparisons with other concurrent validations.

WARNING	NO	YES

This option directs the compiler to not emit warning messages.

FLOAT	SOFTWARE	AUTOMATIC

This option directs the binder to utilize software floating point support.

HISTORY	NO	YES

This option directs the binder to not create tables allowing a stack traceback to be printed if a program terminates from an unhandled exception.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	255
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"

MACRO PARAMETERS

$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	2
$COUNT_LAST	2147483647
$DEFAULT_MEM_SIZE	2**32
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	MC680X0
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	SYSTEM.NULL_ADDRESS
$ENTRY_ADDRESS1	SYSTEM.NULL_ADDRESS+4
$ENTRY_ADDRESS2	SYSTEM.NULL_ADDRESS+8
$FIELD_LAST	255
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100000.0
$GREATER_THAN_DURATION_BASE_LAST	100000000.0
$GREATER_THAN_FLOAT_BASE_LAST	1.80141E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E308

$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.0E308
$HIGH_PRIORITY	16
$ILLEGAL_EXTERNAL_FILE_NAME1	ILLEGAL_EXTERNAL_FILE_NAME1_9012
$ILLEGAL_EXTERNAL_FILE_NAME2	ILLEGAL_EXTERNAL_FILE_NAME2_9012
$INAPPROPRIATE_LINE_LENGTH	-1
$INAPPROPRIATE_PAGE_LENGTH	-1
$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006D1.TST")
$INTEGER_FIRST	-2147483648
$INTEGER_LAST	2147483647
$INTEGER_LAST_PLUS_1	2147483648
$INTERFACE_LANGUAGE	PASCAL
$LESS_THAN_DURATION	-100000.0
$LESS_THAN_DURATION_BASE_FIRST	-100000000.0
$LINE_TERMINATOR	ASCII.CR
$LOW PRIORITY	1
$MACHINE_CODE_STATEMENT	NULL;
$MACHINE_CODE_TYPE	NO_SUCH_TYPE
$MANTISSA_DOC	31
$MAX_DIGITS	15
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2147483648
$MIN_INT	-2147483648

MACRO PARAMETERS

$NAME	SHORT_SHORT_INTEGER
$NAME_LIST	MC680X0
$NAME_SPECIFICATION1	FWB_300:ACVC:RUNNING:X2120A
$NAME_SPECIFICATION2	FWB_300:ACVC:RUNNING:X2120B
$NAME_SPECIFICATION3	FWB_300:ACVC:RUNNING:X3119A
$NEG_BASED_INT	16#F000000E#
$NEW_MEM_SIZE	2**24
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	MC680X0
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	RECORD NULL; END RECORD;
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	1024
$TICK	1.0/60.0
$VARIABLE_ADDRESS	FCNDECL.OBJECT_ADDRESS
$VARIABLE_ADDRESS1	FCNDECL.OBJECT_ADDRESS1
$VARIABLE_ADDRESS2	FCNDECL.OBJECT_ADDRESS2
$YOUR_PRAGMA	EXPORT

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

The complete set of compiler options with their default values is:

```
COMPILE (SOURCE      => ,
         LIBRARY     => ,
         OPTIONS     => (ANNOTATE     => no_value,
                         ERRORS       => 50,
                         LEVEL        => UPDATE,
                         CHECKS       => ALL,
                         GENERICS     => INLINE,
                         FLOAT        => SOFTWARE,
                         MEMORY       => 300),
         DISPLAY     => (OUTPUT       => SCREEN,
                         WARNING      => YES,
                         TEXT         => NO,
                         SHOW         => ALL,
                         DETAIL       => YES,
                         ASSEMBLY     => NONE),
         IMPROVE     => (CALLS        => NORMAL,
                         REDUCTION    => NONE,
                         EXPRESSIONS  => NONE,
                         OBJECT       => PEEPHOLE),
         ALLOCATION  => (STACK        => 2048,
                         GLOBAL       => 1024),
         KEEP        => (TREE         => NO,
                         DEBUG        => NO,
                         COPY         => NO));
```

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

The complete set of binder options with their default values is:

```
BIND (PROGRAM    => ,
      LIBRARY    => ,
      OPTIONS    => (LEVEL        => LINK,
                     COMPONENT    => MPW_TOOL,
                     OBJECT       => AUTOMATIC,
                     UNCALLED     => REMOVE,
                     SLICE        => NO,
                     FLOAT        => AUTOMATIC),
      STACK      => (MAIN         => 16,
                     TASK         => 8,
                     HISTORY      => YES),
      HEAP       => (SIZE         => 32,
                     INCREMENT    => 32),
      INTERFACE  => (DIRECTIVES   => no_value,
                     MODULES      => no_value,
                     SEARCH       => no_value),
      DISPLAY    => (OUTPUT       => SCREEN,
                     DATA         => NONE,
                     WARNING      => YES),
      KEEP       => (DEBUG        => NO));
```

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is
   ..........
   type SHORT_SHORT_INTEGER is range -128 .. 127;
   type SHORT_INTEGER is range    -32_768 .. 32_767;
   type INTEGER is range   -2_147_483_648 .. 2_147_483_647;

   type FLOAT is digits 6 range -2#1.111_1111_1111_1111_1111_1111#E+127 ..
                                 2#1.111_1111_1111_1111_1111_1111#E+127;

   type LONG_FLOAT is digits 15 range
      -(2.0 - 2.0**(-52)) * 2.0**1023 .. +(2.0 - 2.0**(-52)) * 2.0**1023;

   type DURATION is delta 2#0.000_000_000_000_01#
      range -86_400.0 .. 86_400.0;
   ..........
end STANDARD;
```

Alsys Ada Development Environment

APPENDIX F

for the Apple Macintosh Computer

Version 5

Printed: October 1990

TABLE OF CONTENTS

CHAPTER 1

IMPLEMENTATION-DEPENDENT PRAGMAS

1.1 The Pragma INTERFACE

Programs written in Ada can interface with external subprograms written in another language, by use of the pragma INTERFACE. The format of the pragma is:

pragma INTERFACE (*language_name*, *Ada_subprogram_name*);

The *language_name* may be Pascal, C, or Assembler.

The *Ada_subprogram_name* is the name by which the subprogram is known in Ada.

Interfacing the Ada language with other languages is detailed in the *Application Developer's Guide.*

1.2 The Pragma INTERFACE_NAME

To name the external subprogram to which an Ada subprogram is interfaced, as defined in the other language, may require the use of non-Ada naming conventions, such as special characters, or case sensitivity. For this purpose the implementation-dependent pragma INTERFACE_NAME may be used in conjunction with the pragma INTERFACE.

pragma INTERFACE_NAME (*Ada_subprogram_name*, *name_string*);

The *name_string* is a string, which denotes the name of the external subprogram as defined in the other language. The *Ada_subprogram_name* is the name by which the subprogram is known in Ada.

The pragma INTERFACE_NAME may be used anywhere in an Ada program where INTERFACE is allowed (see [13.9]). It must occur after the corresponding pragma INTERFACE and within the same declarative part or package specification.

1.3 The Pragma INLINE

Pragma INLINE is fully supported; however, it is not possible to inline a subprogram in a declarative part.

Note that inlining facilities are also provided by use of the command COMPILE with the option IMPROVE (see the *User's Guide*).

1.4 The Pragma EXPORT

The pragma EXPORT takes a language name and an Ada identifier as arguments. This pragma allows an object defined in Ada to be visible to external programs written in the specified language.

```
pragma EXPORT (language_name, Ada_identifier)
```

Example:

```
package MY_PACKAGE is

    THIS_OBJECT : INTEGER;
    pragma EXPORT (PASCAL, THIS_OBJECT);
    ......

end MY_PACKAGE;
```

Limitations on the use of pragma EXPORT

- This pragma must occur in a declarative part and applies only to objects declared in this same declarative part, that is, generic instantiated objects or renamed objects are excluded.
- The pragma is only to be used for objects with direct allocation mode, which are declared in a library package. The ALSYS implementation gives indirect allocation mode to dynamic objects, and objects that have a significant size (see Section 2.1 of the *Application Developer's Guide*).

1.5 The Pragma EXTERNAL_NAME

To name an exported Ada object as it is identified in the other language may require the use of non-Ada naming conventions, such as special characters, or case sensitivity. For this purpose the implementation-dependent pragma EXTERNAL_NAME may be used in conjunction with the pragma EXPORT:

```
pragma EXTERNAL_NAME (Ada_identifier, name_string);
```

The *name_string* is a string which denotes the name of the identifier defined in the other language. The *Ada_identifier* denotes the exported Ada object.

The pragma EXTERNAL_NAME may be used anywhere in an Ada program where pragma EXPORT is allowed. It must occur after the corresponding pragma EXPORT and within the same library package.

Example:

```
package MY_PACKAGE is

    THIS_OBJECT : INTEGER;
    pragma EXPORT (PASCAL, THIS_OBJECT);
    pragma EXTERNAL_NAME (THIS_OBJECT, "ThisObject");
    ......

end MY_PACKAGE;
```

1.6 The Pragma INDENT

This pragma is only used by AdaReformat. This tool offers the functionalities of a pretty-printer in an Ada environment.

The pragma is placed in the source file and interpreted by AdaReformat.

pragma INDENT(OFF) causes AdaReformat not to modify the source lines after this pragma.

pragma INDENT(ON) causes AdaReformat to resume its action after this pragma.

1.7 The Pragma IMPROVE

This pragma is used to suppress implicit components from a record type.

pragma IMPROVE (TIME | SPACE, [ON =>] simple_name);

See Section 4.8 for the complete description.

1.8 The other Pragmas

Pragma PACK is discussed in detail in the section on representation clauses and records (Chapter 4).

Pragma PRIORITY is accepted with the range of priorities running from 1 to 16 (see the definition of the predefined package SYSTEM in Section 3). Undefined priority (no pragma PRIORITY) is treated as though it were less than every defined priority value.

In addition to pragma SUPPRESS, it is possible to suppress all checks in a given compilation by the use of the Compiler option CHECKS. (See Chapter 4 of the *User's Guide*.)

1.9 Pragmas with no Effect

The following pragmas have no effect:

CONTROLLED
MEMORY_SIZE
STORAGE_UNIT
SYSTEM_NAME
OPTIMIZE

For optimization, certain facilities are provided through use of the command COMPILE with the option IMPROVE (see the *User's Guide*).

CHAPTER 2

IMPLEMENTATION-DEPENDENT ATTRIBUTES

2.1 Attributes used in Record Representation Clauses

In addition to the Representation Attributes of [13.7.2] and [13.7.3], the following five attributes are used to form names of indirect and implicit components for use in record representation clauses, as described in Section 4.8.

'OFFSET
'RECORD_SIZE
'VARIANT_INDEX
'ARRAY_DESCRIPTOR
'RECORD_DESCRIPTOR

2.2 Limitations on the use of the Attribute ADDRESS

The attribute ADDRESS is implemented for all prefixes that have meaningful addresses.

Note that the value returned by the attribute ADDRESS is undefined before the elaboration of the subprogram body (when 'ADDRESS is applied to a subprogram).

The following entities do not have meaningful addresses and will therefore cause a compilation error if used as prefix to ADDRESS:

- A constant that is implemented as an immediate value, i.e., does not have any space allocated for it
- A package specification that is not a library unit
- A package body that is not a library unit or subunit
- A function that renames an enumeration literal.

2.3 The Attribute IMPORT

This attribute is a function which takes two literal strings as arguments; the first one denotes a language name and the second one denotes an external symbol name. It yields the address of this external symbol. The prefix of this attribute must be SYSTEM.ADDRESS. The value of this attribute is of the type SYSTEM.ADDRESS. The syntax is:

SYSTEM.ADDRESS'IMPORT ("*Language_name*", "*external_symbol_name*")

Following are two examples which illustrate the use of this attribute. Note that in these examples the sizes of the predefined integer types SHORT_INTEGER and INTEGER are respectively 16 and 32 bits as defined in package STANDARD (See Chapter 3).

Example 1:

SYSTEM.ADDRESS'IMPORT is used in an address clause in order to access a global object defined in a C library:

For the language C:

```
extern int errno;
.....
```

For the language Ada:

```
package MY_PACK is

    ERROR_NO : INTEGER;
    for ERROR_NO use at SYSTEM.ADDRESS'IMPORT ("C", "errno");
    ....

end MY_PACK;
```

Note that implicit initializations are performed on the declaration of objects; objects of type access are implicitly initialized to null.

Example 2:

The second example shows another use of 'IMPORT which avoids implicit initializations.

SYSTEM.ADDRESS'IMPORT is used in a renaming declaration to give a new name to an external object:

For the language C:

```
struct record_c {
    short i1;
    short i2;
} rec;
```

For the language Ada:

```
type RECORD_C is
    record
        I1 : SHORT_INTEGER;
        I2 : SHORT_INTEGER;
    end record;

type ACCESS_RECORD is access RECORD_C;
function CONVERT_TO_ACCESS_RECORD is new
        UNCHECKED_CONVERSION (SYSTEM.ADDRESS, ACCESS_RECORD);
X: RECORD_C renames CONVERT_TO_ACCESS_RECORD
        (SYSTEM.ADDRESS'IMPORT("C", "rec") ).all;
```

In this example, no implicit initialization is done on the renamed object X.

Note that the object is actually defined in the external world and is only *referenced* in the Ada world.

CHAPTER 3

THE PACKAGES SYSTEM AND STANDARD

This section contains information on two predefined library packages:

- a complete listing of the visible part of the specification of the package SYSTEM
- a list of the implementation-dependent declarations in the package STANDARD.

The package SYSTEM

```
package SYSTEM is

   type ADDRESS is private;

   type NAME is (MC680X0);

   SYSTEM_NAME  : constant NAME := MC680X0;

   STORAGE_UNIT : constant := 8;
   MEMORY_SIZE  : constant := 2**32;

   MIN_INT      : constant := - (2**31);
   MAX_INT      : constant := 2**31 - 1;
   MAX_DIGITS   : constant := 15;
   MAX_MANTISSA : constant := 31;
   FINE_DELTA   : constant := 2#1.0#E-31;
   TICK         : constant := 1.0/60.0;

   subtype PRIORITY is INTEGER range 1 .. 16;

   NULL_ADDRESS : constant ADDRESS;

   function VALUE (LEFT : in STRING) return ADDRESS;
```

```
      ADDRESS_WIDTH : constant := 8;
      subtype ADDRESS_STRING is STRING(1..ADDRESS_WIDTH);

      function IMAGE (LEFT : in ADDRESS) return ADDRESS_STRING;

      type OFFSET is range -2**31 .. 2**31-1;
         -- This type designates a number of storage units (bytes).

      function "+" (LEFT : in ADDRESS; RIGHT : in OFFSET)  return ADDRESS;
      function "+" (LEFT : in OFFSET;  RIGHT : in ADDRESS) return ADDRESS;
      function "-" (LEFT : in ADDRESS; RIGHT : in OFFSET)  return ADDRESS;
      function "-" (LEFT : in ADDRESS; RIGHT : in ADDRESS) return OFFSET;

      function "<=" (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
      function "<"  (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
      function ">=" (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
      function ">"  (LEFT, RIGHT : in ADDRESS) return BOOLEAN;

      function "mod" (LEFT : in ADDRESS; RIGHT : in POSITIVE) return NATURAL;

      type ROUND_DIRECTION is (DOWN, UP);

      function ROUND (VALUE     : in ADDRESS;
                      DIRECTION : in ROUND_DIRECTION;
                      MODULUS   : in POSITIVE) return ADDRESS;

      generic
         type TARGET is private;
      function FETCH_FROM_ADDRESS (A : in ADDRESS) return TARGET;
      generic
         type TARGET is private;
      procedure ASSIGN_TO_ADDRESS (A : in ADDRESS; T : in TARGET);


   type OBJECT_LENGTH is range 0 .. 2**31 -1;
        -- This type designates the size of an object in storage units.
   procedure MOVE (TO     : in ADDRESS;
                   FROM   : in ADDRESS;
                   LENGTH : in OBJECT_LENGTH);

private

   -- Private part of the SYSTEM package.

end SYSTEM;
```

The package STANDARD

The following are the implementation-dependent declarations in the package STANDARD:

```
type SHORT_SHORT_INTEGER    is range -2**7 .. 2**7 -1;
type SHORT_INTEGER          is range -2**15 .. 2**15 -1;
type INTEGER                is range  -2**31 .. 2**31 -1;

type FLOAT is digits 6 range
   -(2.0 - 2.0**(-23)) * 2.0**127 ..
   +(2.0 - 2.0**(-23)) * 2.0**127;

type LONG_FLOAT is digits 15 range
   -(2.0 - 2.0**(-52)) * 2.0**1023 ..
   +(2.0 - 2.0**(-52)) * 2.0**1023;

type DURATION is delta 2.0**(-14) range -86_400.0 .. 86_400.0;
```

CHAPTER 4

TYPE REPRESENTATION CLAUSES

The aim of this section is to explain how objects are represented and allocated by the Alsys Ada compiler for MC680X0 machines and how it is possible to control this using representation clauses.

The representation of an object is closely connected with its type. For this reason this section addresses successively the representation of enumeration, integer, floating point, fixed point, access, task, array and record types. For each class of type the representation of the corresponding objects is described.

Except in the case of array and record types, the description for each class of type is independent of the others. To understand the representation of an array type it is *necessary to understand first the representation of its components.* The same rule applies to record types.

Apart from implementation defined pragmas, Ada provides three means to control the size of objects:

- a (predefined) pragma PACK, when the object is an array, an array component, a record or a record component
- a record representation clause, when the object is a record or a record component
- a size specification, in any case.

For each class of types the effect of a size specification alone is described. Interference between size specifications, packing and record representation clauses is described under array and record types.

4.1 Enumeration Types

Internal codes of enumeration literals

When no enumeration representation clause applies to an enumeration type, the internal code associated with an enumeration literal is the position number of the enumeration literal. Then, for an enumeration type with n elements, the internal codes are the integers 0, 1, 2, .. , n-1.

An enumeration representation clause can be provided to specify the value of each internal code as described in [13.3]. The Alsys compiler fully implements enumeration representation clauses.

As internal codes must be machine integers the internal codes provided by an enumeration representation clause must be in the range -2^{31} .. $2^{31}-1$.

Encoding of enumeration values

An enumeration value is always represented by its internal code in the program generated by the compiler.

When an enumeration type is not a boolean type or is a boolean type with an enumeration representation clause, binary code is used to represent internal codes. Negative codes are then represented using two's complement.

When a boolean type has no enumeration representation clause, the internal code 0 is represented by a succession of 0s and the internal code 1 is represented by a succession of 1s. The length of this pattern of 0s or of 1s is the size of the boolean value.

Minimum size of an enumeration subtype

The minimum size of an enumeration subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form.

For a static subtype, if it has a null range its minimum size is 1. Otherwise, if m and M are the values of the internal codes associated with the first and last enumeration values of the subtype, then its minimum size L is determined as follows. For $m >= 0$, L is the smallest positive integer such that $M <= 2^{L}-1$. For $m < 0$, L is the smallest positive integer such that $-2^{L-1} <= m$ and $M <= 2^{L-1}-1$.

```
type COLOR is (GREEN, BLACK, WHITE, RED, BLUE, YELLOW);
-- The minimum size of COLOR is 3 bits.

subtype BLACK_AND_WHITE is COLOR range BLACK .. WHITE;
-- The minimum size of BLACK_AND_WHITE is 2 bits.

subtype BLACK_OR_WHITE is BLACK_AND_WHITE range X .. X;
-- Assuming that X is not static, the minimum size of BLACK_OR_WHITE is
-- 2 bits (the same as the minimum size of its type mark BLACK_AND_WHITE).
```

Size of an enumeration subtype

When no size specification is applied to an enumeration type or first named subtype, the objects of that type or first named subtype are represented as signed machine integers. The machine provides 8, 16 and 32 bit integers, and the compiler selects automatically the smallest signed machine integer which can hold each of the internal codes of the enumeration type. The size of the enumeration type and of any of its subtypes is thus 8, 16 or 32 bits.

When a size specification is applied to an enumeration type, this enumeration type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies:

```
type EXTENDED is
  ( -- The usual American ASCII characters.
      NUL,    SOH,    STX,    ETX,    EOT,    ENQ,    ACK,    BEL,
      BS,     HT,     LF,     VT,     FF,     CR,     SO,     SI,
      DLE,    DC1,    DC2,    DC3,    DC4,    NAK,    SYN,    ETB,
      CAN,    EM,     SUB,    ESC,    FS,     GS,     RS,     US,
      ' ',    '!',    '"',    '#',    '$',    '%',    '&',    ''',
      '(',    ')',    '*',    '+',    ',',    '-',    '.',    '/',
      '0',    '1',    '2',    '3',    '4',    '5',    '6',    '7',
      '8',    '9',    ':',    ';',    '<',    '=',    '>',    '?',
      '@',    'A',    'B',    'C',    'D',    'E',    'F',    'G',
      'H',    'I',    'J',    'K',    'L',    'M',    'N',    'O',
      'P',    'Q',    'R',    'S',    'T',    'U',    'V',    'W',
      'X',    'Y',    'Z',    '[',    '\',    ']',    '^',    '_',
      '`',    'a',    'b',    'c',    'd',    'e',    'f',    'g',
      'h',    'i',    'j',    'k',    'l',    'm',    'n',    'o',
      'p',    'q',    'r',    's',    't',    'u',    'v',    'w',
      'x',    'y',    'z',    '{',    '|',    '}',    '~',    DEL,
      -- Extended characters
      LEFT_ARROW,
      RIGHT_ARROW,
      UPPER_ARROW,
      LOWER_ARROW,
      UPPER_LEFT_CORNER,
      UPPER_RIGHT_CORNER,
      LOWER_RIGHT_CORNER,
      LOWER_LEFT_CORNER);

for EXTENDED'SIZE use 8;
-- The size of type EXTENDED will be one byte. Its objects will be represented
-- as unsigned 8 bit integers.
```

The Alsys compiler fully implements size specifications. Nevertheless, as enumeration values are coded using integers, the specified length cannot be greater than 32 bits.

Size of the objects of an enumeration subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of an enumeration subtype has the same size as its subtype.

Alignment of an enumeration subtype

An enumeration subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, it is otherwise even byte aligned.

Address of an object of an enumeration subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an enumeration subtype is even when its subtype is even byte aligned.

4.2 Integer Types

Predefined integer types

There are three predefined integer types in the Alsys implementation for MC680X0 machines:

```
type SHORT_SHORT_INTEGER     is range -2**7 .. 2**7 -1;
type SHORT_INTEGER           is range -2**15 .. 2**15 -1;
type INTEGER                 is range -2**31 .. 2**31 -1;
```

Selection of the parent of an integer type

An integer type declared by a declaration of the form:

```
type T is range L .. R;
```

is implicitly derived from a predefined integer type. The compiler automatically selects the predefined integer type whose range is the shortest that contains the valιes L to R inclusive.

Encoding of integer values

Binary code is used to represent integer values. Negative numbers are represented using two's complement.

Minimum size of an integer subtype

The minimum size of an integer subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form.

For a static subtype, if it has a null range its minimum size is 1. Otherwise, if m and M are the lower and upper bounds of the subtype, then its minimum size L is determined as follows. For $m >= 0$, L is the smallest positive integer such that $M <= 2^L-1$. For $m < 0$, L is the smallest positive integer that $-2^{L-1} <= m$ and $M <= 2^{L-1}-1$.

```
subtype S is INTEGER range 0 .. 7;
-- The minimum size of S is 3 bits.

subtype D is S range X .. Y;
-- Assuming that X and Y are not static, the minimum size of
-- D is 3 bits (the same as the minimum size of its type mark S).
```

Size of an integer subtype

The sizes of the predefined integer types SHORT_SHORT_INTEGER, SHORT_INTEGER and INTEGER are respectively 8, 16 and 32 bits.

When no size specification is applied to an integer type or to its first named subtype (if any), its size and the size of any of its subtypes is the size of the predefined type from which it derives, directly or indirectly. For example:

```
type S is range 80 .. 100;
-- S is derived from the predefined 8 bit integer, its size is 8 bits.

type J is range 0 .. 255;
-- J is derived from the predefined 16 bit integer, its size is 16 bits.

type N is new J range 80 .. 100;
-- N is indirectly derived from the predefined 16 bit integer, its size
-- is 16 bits.
```

When a size specification is applied to an integer type, this integer type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies:

```
type S is range 80 .. 100;
for S'SIZE use 32;
-- S is derived from an 8 bit integer, but its size is 32 bits
-- because of the size specification.

type J is range 0 .. 255;
for J'SIZE use 8;
-- J is derived from a 16 bit integer, but its size is 8 bits because
-- of the size specification.

type N is new J range 80 .. 100;
-- N is indirectly derived from a 16 bit integer, but its size is 8 bits
-- because N inherits the size specification of J.
```

The Alsys compiler fully implements size specifications. Nevertheless, as integers are implemented using machine integers, the specified length cannot be greater than 32 bits.

Size of the objects of an integer subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of an integer subtype has the same size as its subtype.

Alignment of an integer subtype

An integer subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, it is otherwise even byte aligned.

Address of an object of an integer subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an integer subtype is even when its subtype is even byte aligned.

4.3 Floating Point Types

Predefined floating point types

There are two predefined floating point types in the Alsys implementation for MC680X0 machines:

```
type FLOAT is
    digits 6 range -(2.0 - 2.0**(-23))*2.0**127 .. (2.0 - 2.0**(-23))*2.0**127;

type LONG_FLOAT is
    digits 15 range -(2.0 - 2.0**(-51))*2.0**1023 .. (2.0 - 2.0**(-51))*2.0**1023;
```

Selection of the parent of a floating point type

A floating point type declared by a declaration of the form:

```
type T is digits D [range L .. R];
```

is implicitly derived from a predefined floating point type. The compiler automatically selects the smallest predefined floating point type whose number of digits is greater than or equal to D and which contains the values L to R inclusive.

Encoding of floating point values

In the program generated by the compiler, floating point values are represented using the IEEE standard formats for single and double floats.

The values of the predefined type FLOAT are represented using the single float format. The values of the predefined type LONG_FLOAT are represented using the double float format. The values of any other floating point type are represented in the same way as the values of the predefined type from which it derives, directly or indirectly.

Minimum size of a floating point subtype

The minimum size of a floating point subtype is 32 bits if its base type is FLOAT or a type derived from FLOAT; it is 64 bits if its base type is LONG_FLOAT or a type derived from LONG_FLOAT.

Size of a floating point subtype

The sizes of the predefined floating point types FLOAT and LONG_FLOAT are respectively 32 and 64 bits.

The size of a floating point type and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly.

The only size that can be specified for a floating point type or first named subtype using a size specification is its usual size (32 or 64 bits).

Size of the objects of a floating point subtype

An object of a floating point subtype has the same size as its subtype.

Alignment of a floating point subtype

A floating point subtype is always even byte aligned.

Address of an object of a floating point subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a floating point subtype is always even, since its subtype is even byte aligned.

4.4 Fixed Point Types

Small of a fixed point type

If no specification of small applies to a fixed point type, then the value of small is determined by the value of delta as defined by [3.5.9].

A specification of small can be used to impose a value of small. The value of small is required to be a power of two.

Predefined fixed point types

To implement fixed point types, the Alsys compiler for MC680X0 machines uses a set of anonymous predefined types of the form:

```
type FIXED_8     is delta D range (-2**07-1)*S .. 2**07*S;
for FIXED_8'SMALL use S;

type FIXED_16    is delta D range (-2**15-1)*S .. 2**15*S;
for FIXED_16'SMALL use S;

type FIXED_32    is delta D range (-2**31-1)*S .. 2**31*S;
for FIXED_32'SMALL use S;
```

where D is any real value and S any power of two less than or equal to D.

Selection of the parent of a fixed point type

A fixed point type declared by a declaration of the form:

type T **is delta** D **range** L .. R;

possibly with a specification of small:

for T'SMALL **use** S;

is implicitly derived from a predefined fixed point type. The compiler automatically selects the predefined fixed point type whose small and delta are the same as thc small and delta of T and whose range is the shortest that includes the values L to R inclusive.

Encoding of fixed point values

In the program generated by the compiler, a safe value V of a fixed point subtype F is represented as the integer:

V / F'BASE'SMALL

Minimum size of a fixed point subtype

The minimum size of a fixed point subtype is the minimum number of binary digits that is necessary for representing the values of the range of the subtype using the small of the base type.

For a static subtype, if it has a null range its minimum size is 1. Otherwise, s and S being the bounds of the subtype, if i and I are the integer representations of m and M, the smallest and the greatest model numbers of the base type such that s < m and M < S, then the minimum size L is determined as follows. For i >= 0, L is the smallest positive integer such that $I <= 2^{L}-1$. For i < 0, L is the smallest positive integer such that $-2^{L-1} <= i$ and $I <= 2^{L-1}-1$.

```
type F is delta 2.0 range 0.0 .. 500.0;
-- The minimum size of F is 8 bits.

subtype S is F delta 16.0 range 0.0 .. 250.0;
-- The minimum size of S is 7 bits.

subtype D is S range X .. Y;
-- Assuming that X and Y are not static, the minimum size  of D is 7 bits
-- (the same as the minimum size of its type mark S).
```

Size of a fixed point subtype

The sizes of the predefined fixed point types FIXED_8, FIXED_16 and FIXED_32 are respectively 8, 16 and 32 bits.

When no size specification is applied to a fixed point type or to its first named subtype, its size and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly. For example:

```
type S is delta 0.01 range 0.8 .. 1.0;
-- S is derived from an 8 bit predefined fixed type, its size is 8 bits.

type F is delta 0.01 range 0.0 .. 2.0;
-- F is derived from a 16 bit predefined fixed type, its size is 16 bits.

type N is new F range 0.8 .. 1.0;
-- N is indirectly derived from a 16 bit predefined fixed type, its size is 16 bits.
```

When a size specification is applied to a fixed point type, this fixed point type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies:

```
type S is delta 0.01 range 0.8 .. 1.0;
for S'SIZE use 32;
-- S is derived from an 8 bit predefined fixed type, but its size is 32 bits
-- because of the size specification.
```

```
type F is delta 0.01 range 0.0 .. 2.0;
for F'SIZE use 8;
-- F is derived from a 16 bit predefined fixed type, but its size is 8 bits
-- because of the size specification.

type N is new F range 0.8 .. 1.0;
-- N is indirectly derived from a 16 bit predefined fixed type, but its size is
-- 8 bits because N inherits the size specification of F.
```

The Alsys compiler fully implements size specifications. Nevertheless, as fixed point objects are represented using machine integers, the specified length cannot be greater than 32 bits.

Size of the objects of a fixed point subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of a fixed point type has the same size as its subtype.

Alignment of a fixed point subtype

A fixed point subtype is byte aligned if its size is less than or equal to 8 bits, and is otherwise even byte aligned.

Address of an object of a fixed point subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a fixed point subtype is even when its subtype is even byte aligned.

4.5 Access Types

Collection Size

When no specification of collection size applies to an access type, no storage space is reserved for its collection, and the value of the attributc STORAGE_SIZE is then 0.

As described in [13.2], a specification of collection size can be provided in order to reserve storage space for the collection of an access type. The Alsys compiler fully implements this kind of specification.

Encoding of access values.

Access values are machine addresses.

Minimum size of an access subtype

The minimum size of an access subtype is 32 bits.

Size of an access subtype

The size of an access subtype is 32 bits, the same as its minimum size.

The only size that can be specified for an access type using a size specification is its usual size (32 bits).

Size of an object of an access subtype

An object of an access subtype has the same size as its subtype, thus an object of an access subtype is always 32 bits long.

Alignment of an access subtype

An access subtype is always even byte aligned.

Address of an object of an access subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an access subtype is always even, since its subtype is even byte aligned.

4.6 Task Types

Storage for a task activation

When no length clause is used to specify the storage space to be reserved for a task activation, the storage space indicated at bind time is used for this activation.

As described in [13.2], a length clause can be used to specify the storage space for the activation of each of the tasks of a given type. In this case the value indicated at bind time is ignored for this task type, and the length clause is obeyed.

Encoding of task values

Encoding of a task value is not described here.

Minimum size of a task subtype

The minimum size of a task subtype is 32 bits.

Size of a task subtype

The size of a task subtype is 32 bits, the same as its minimum size.

A size specification has no effect on a task type. The only size that can be specified using such a length clause is its minimum size.

Size of the objects of a task subtype

An object of a task subtype has the same size as its subtype. Thus an object of a task subtype is always 32 bits long.

Alignment of a task subtype

A task subtype is always even byte aligned.

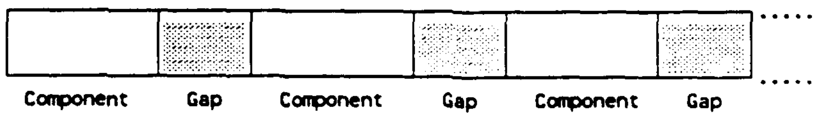
Address of an object of a task subtype

Provided its alignment is not constrained by a record representation clause, the address of an object of a task subtype is always even, since its subtype is even byte aligned.

4.7 Array Types

Layout of an array

Each array is allocated in a contiguous area of storage units. All the components have the same size. A gap may exist between two consecutive components (and after the last one). All the gaps have the same size.



- *Components*

If the array is not packed, the size of the components is the size of the subtype of the components:

```
type A is array (1 .. 8) of BOOLEAN;
-- The size of the components of A is the size of the type BOOLEAN: 8 bits.

type DECIMAL_DIGIT is range 0 .. 9;
for DECIMAL_DIGIT'SIZE use 4;

type BINARY_CODED_DECIMAL is
    array (INTEGER range <>) of DECIMAL_DIGIT;
-- The size of the type DECIMAL_DIGIT is 4 bits. Thus in an array of
-- type BINARY_CODED_DECIMAL each component will be represented on
-- 4 bits as in the usual BCD representation.
```

If the array is packed and its components are neither records nor arrays, the size of the components is the minimum size of the subtype of the components:

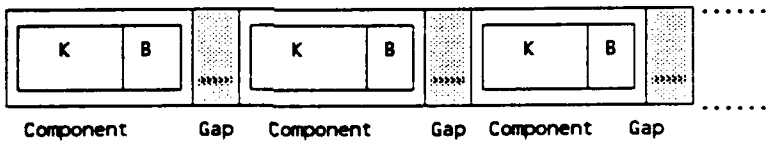
```
type A is array (1 .. 8) of BOOLEAN;
pragma PACK(A);
-- The size of the components of A is the minimum size of the type BOOLEAN:
-- 1 bit.

type DECIMAL_DIGIT is range 0 .. 9;
for DECIMAL_DIGIT'SIZE use 32;
type BINARY_CODED_DECIMAL is
    array (INTEGER range <>) of DECIMAL_DIGIT;
pragma PACK(BINARY_CODED_DECIMAL);
-- The size of the type DECIMAL_DIGIT is 32 bits, but, as
-- BINARY_CODED_DECIMAL is packed, each component of an array of this
-- type will be represented on 4 bits as in the usual BCD representation.
```

Packing the array has no effect on the size of the components when the components are records or arrays.

■ *Gaps*

If the components are records or arrays, no size specification applies to the subtype of the components and the array is not packed, then the compiler may choose a representation with a gap after each component; the aim of the insertion of such gaps is to optimize access to the array components and to their subcomponents. The size of the gap is chosen so that the relative displacement of consecutive components is a multiple of the alignment of the subtype of the components. This strategy allows each component and subcomponent to have an address consistent with the alignment of its subtype:

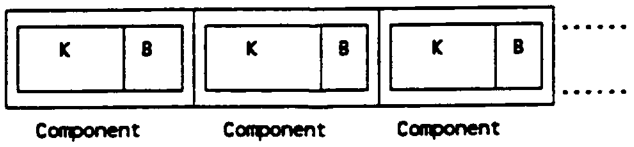
```
type R is
    record
        K : SHORT_INTEGER;  -- integer is even byte aligned.
        B : BOOLEAN;  -- BOOLEAN is byte aligned.
    end record;
-- Record type R is even byte aligned. Its size is 24 bits.

type A is array (1 .. 10) of R;
-- A gap of one byte is inserted  after each component in order to respect the
-- alignment of type R. The size of an array of type A will be 320 bits.
```



Array of type A: each subcomponent K has an even offset.

If a size specification applies to the subtype of the components or if the array is packed, no gaps are inserted:

```
type R is
    record
        K : SHORT_INTEGER;
        B : BOOLEAN;
    end record;

type A is array (1 .. 10) of R;
pragma PACK(A);
-- There is no gap in an array of type A because
-- A is packed.
-- The size of an object of type A will be 240 bits.

type NR is new R;
for NR'SIZE use 24;

type B is array (1 .. 10) of NR;
-- There is no gap in an array of type B because
-- NR has a size specification.
-- The size of an object of type B will be 240 bits.
```



Array of type A or B: a subcomponent K can have an odd offset.

Size of an array subtype

The size of an array subtype is obtained by multiplying the number of its components by the sum of the size of the components and the size of the gaps (if any). If the subtype is unconstrained the maximum number of components is considered.

The size of an array subtype cannot be computed at compile time

- if it has non-static constraints or is an unconstrained array type with non-static index subtypes (because the number of components can then only be determined at run time).
- if the components are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static (because the size of the components and the size of the gaps can then only be determined at run time).

As has been indicated above, the effect of a pragma PACK on an array type is to suppress the gaps and to reduce the size of the components. The consequence of packing an array type is thus to reduce its size.

If the components of an array are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static, the compiler ignores any pragma PACK applied to the array type but issues a warning message. Apart from this limitation, array packing is fully implemented by the Alsys compiler.

A size specification applied to an array type or first named subtype has no effect. The only size that can be specified using such a length clause is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of an array is as expected by the application.

Size of the objects of an array subtype

The size of an object of an array subtype is always equal to the size of the subtype of the object.

Alignment of an array subtype

If no pragma PACK applies to an array subtype and no size specification applies to its components, the array subtype is even byte aligned if the subtype of its components is even byte aligned. Otherwise it is byte aligned.

If a pragma PACK applies to an array subtype or if a size specification applies to its components (so that there are no gaps), the alignment of the array subtype is as given in the following table:

		relative displacement of components		
		even number of bytes	odd number of bytes	not a whole number of bytes
Component subtype alignment	even byte	even byte	byte	bit
	byte	byte	byte	bit
	bit	bit	bit	bit

Address of an object of an array subtype

Provided its alignment is not constrained by a record representation clause, the address of an object of an array subtype is even when its subtype is even byte aligned.

4.8 Record Types

Layout of a record

Each record is allocated in a contiguous area of storage units. The size of a record component depends on its type. Gaps may exist between some components.

The positions and the sizes of the components of a record type object can be controlled using a record representation clause as described in [13.4]. In the Alsys implementation for MC680X0 machines there is no restriction on the position that can be specified for a component of a record. If a component is of an enumeration, integer or fixed point type, its size can be any size from the minimum size of its subtype to 32 bits. If a component is of another class of type, its size must be the size of its subtype.

```
type INTERRUPT_MASK is array (0 .. 2) of BOOLEAN;
pragma PACK(INTERRUPT_MASK);
-- The size of INTERRUPT_MASK is 3 bits.

type CONDITION_CODE is 0 .. 1;
-- The size of CONDITION_CODE is 8 bits, its minimum size is 1 bit.

type STATUS_BIT is new BOOLEAN;
for STATUS_BIT'SIZE use 1;
-- The size and the minimum size of STATUS_BIT are 1 bit.

SYSTEM    : constant := 0;
USER      : constant := 1;

type STATUS_REGISTER is
     record
          T    : STATUS_BIT;             -- Trace
          S    : STATUS_BIT;             -- Supervisor
          I    : INTERRUPT_MASK;         -- Interrupt mask
          X    : CONDITION_CODE;         -- Extend
          N    : CONDITION_CODE;         -- Negative
          Z    : CONDITION_CODE;         -- Zero
          V    : CONDITION_CODE;         -- Overflow
          C    : CONDITION_CODE;         -- Carry
     end record;
-- This type can be used to map the status register of a MC68000 processor:

for STATUS_REGISTER use
     record at mod 2;
          T    at SYSTEM   range 0 .. 0;
          S    at SYSTEM   range 2 .. 2;
          I    at SYSTEM   range 5 .. 7;
          X    at USER     range 3 .. 3;
          N    at USER     range 4 .. 4;
          Z    at USER     range 5 .. 5;
          V    at USER     range 6 .. 6;
          C    at USER     range 7 .. 7;
     end record;
```

WARNING: Note that bits are numbered from the high order end of a byte. See Chapter 1 of the *Application Developer's Guide* for a discussion of bit numbering.

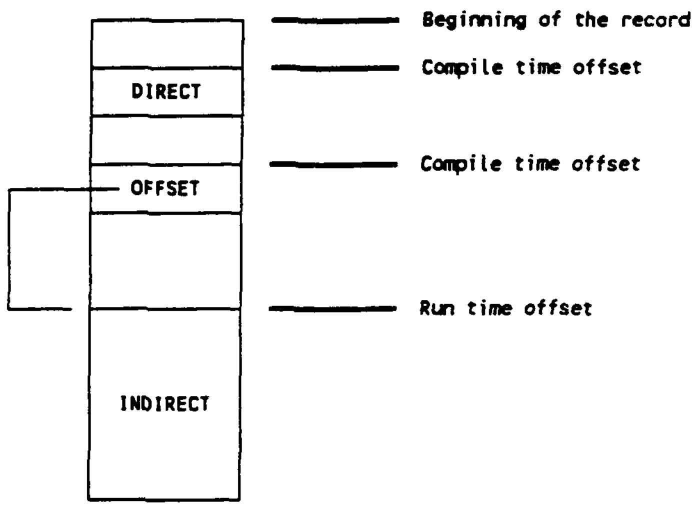
A record representation clause need not specify the position and the size for every component.

If no component clause applies to a component of a record, its size is the size of its subtype. Its position is chosen by the compiler so as to optimize access to the components of the record: the offset of the component is chosen as a multiple of 8 bits if the objects of the component subtype are usually byte aligned, but a multiple of 16 bits if these objects are usually even byte aligned. Moreover, the compiler chooses the position of the component so as to reduce the number of gaps and thus the size of the record objects.

Because of these optimizations, there is no connection between the order of the components in a record type declaration and the positions chosen by the compiler for the components in a record object.

Indirect components

If the offset of a component cannot be computed at compile time, this offset is stored in the record objects at run time and used to access the component. Such a component is said to be indirect while other components are said to be direct:



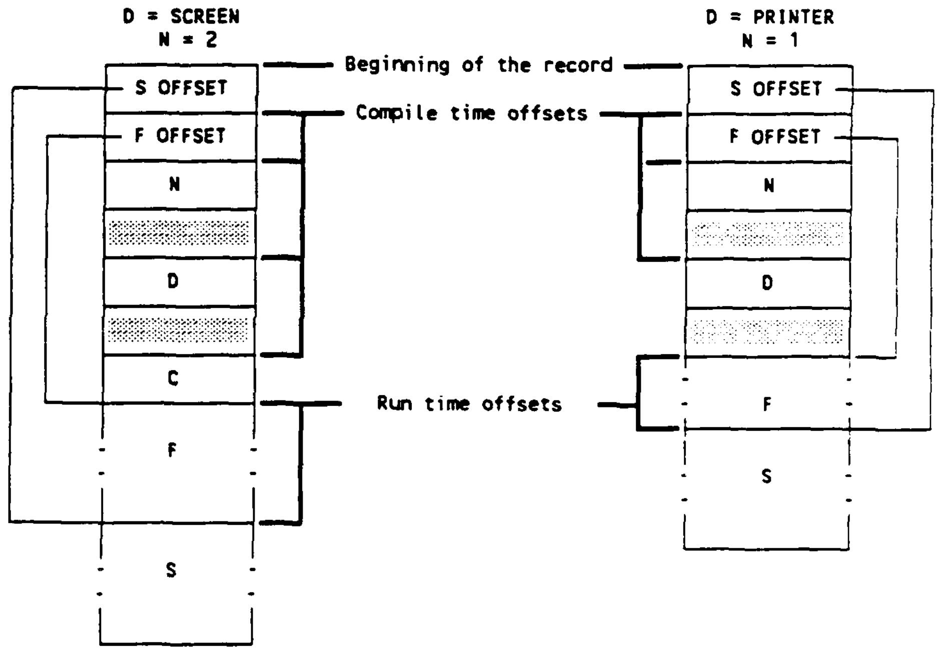
A direct and an indirect component

If a record component is a record or an array, the size of its subtype may be evaluated at run time and may even depend on the discriminants of the record. We will call these components dynamic components:

```
type DEVICE is (SCREEN, PRINTER);

type COLOR is (GREEN, RED, BLUE);

type SERIES is array (POSITIVE range < >) of INTEGER;

type GRAPH (L : NATURAL) is
    record
        X : SERIES(1 .. L); -- The size of X depends on L
        Y : SERIES(1 .. L); -- The size of Y depends on L
    end record;

Q : POSITIVE;
```

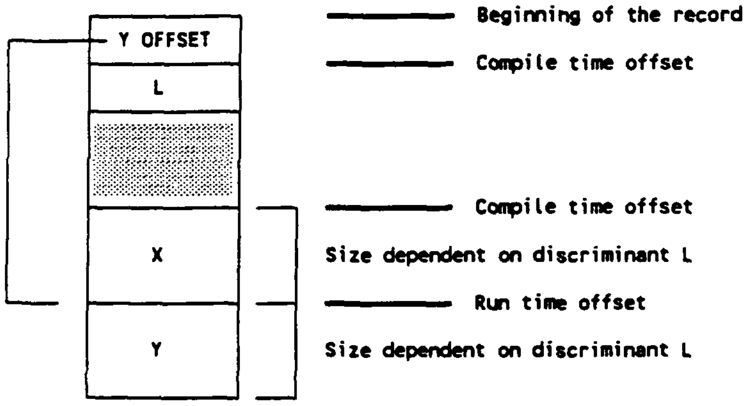
```
type PICTURE (N : NATURAL; D : DEVICE) is
    record
        F : GRAPH(N); -- The size of F depends on N
        S : GRAPH(Q); -- The size of S depends on Q
        case D is
            when SCREEN =>
                C : COLOR;
            when PRINTER =>
                null;
        end case;
    end record;
```

Any component placed after a dynamic component has an offset which cannot be evaluated at compile time and is thus indirect. In order to minimize the number of indirect components, the compiler groups the dynamic components together and places them at the end of the record:



The record type PICTURE: F and S are placed at the end of the record

Thanks to this strategy, the only indirect components are dynamic components. But not all dynamic components are necessarily indirect: if there are dynamic components in a component list which is not followed by a variant part, then exactly one dynamic component of this list is a direct component because its offset can be computed at compilation time (the only dynamic components that are direct components are in this situation):



The record type GRAPH: the dynamic component X is a direct component.

The offset of an indirect component is always expressed in storage units.

The space reserved for the offset of an indirect component must be large enough to store the size of any value of the record type (the maximum potential offset). The compiler evaluates an upper bound MS of this size and treats an offset as a component having an aronymous integer type whose range is 0 .. MS.

If C is the name of an indirect component, then the offset of this component can be denoted in a component clause by the implementation generated name C'OFFSET.

Implicit components

In some circumstances, access to an object of a record type or to its components involves computing information which only depends on the discriminant values. To avoid useless recomputation the compiler stores this information in the record objects, updates it when the values of the discriminants are modified and uses it when the objects or its components are accessed. This information is stored in special components called implicit components.

An implicit component may contain information which is used when the record object or several of its components are accessed. In this case the component will be included in any record object (the implicit component is considered to be declared before any variant part in the record type declaration). There can be two components of this kind; one is called RECORD_SIZE and the other VARIANT_INDEX.

On the other hand an implicit component may be used to access a given record component. In that case the implicit component exists whenever the record component exists (the implicit component is considered to be declared at the same place as the record component). Components of this kind are called ARRAY_DESCRIPTORs or RECORD_DESCRIPTORs.

- *RECORD_SIZE*

This implicit component is created by the compiler when the record type has a variant part and its discriminants are defaulted. It contains the size of the storage space necessary to store the current value of the record object (note that the storage effectively allocated for the record object may be more than this).

The value of a RECORD_SIZE component may denote a number of bits or a number of storage units. In general it denotes a number of storage units, but if any component clause specifies that a component of the record type has an offset or a size which cannot be expressed using storage units, then the value designates a number of bits.

The implicit component RECORD_SIZE must be large enough to store the maximum size of any value of the record type. The compiler evaluates an upper bound MS of this size and then considers the implicit component as having an anonymous integer type whose range is 0 .. MS.

If R is the name of the record type, this implicit component can be denoted in a component clause by the implementation generated name R'RECORD_SIZE.

- *VARIANT_INDEX*

This implicit component is created by the compiler when the record type has a variant part. It indicates the set of components that are present in a record value. It is used when a discriminant check is to be done.

Component lists that do not contain a variant part are numbered. These numbers are the possible values of the implicit component VARIANT_INDEX.

```
type VEHICLE is (AIRCRAFT, ROCKET, BOAT, CAR);

type DESCRIPTION (KIND : VEHICLE := CAR) is
    record
        SPEED : INTEGER;
        case KIND is
            when AIRCRAFT | CAR =>
                WHEELS : INTEGER;
                case KIND is
                    when AIRCRAFT =>        -- 1
                       WINGSPAN : INTEGER;
                    when others =>          -- 2
                       null;
                end case;
            when BOAT =>      -- 3
                STEAM  : BOOLEAN;
            when ROCKET =>    -- 4
                STAGES : INTEGER;
        end case;
    end record;
```

The value of the variant index indicates the set of components that are present in a record value:

Variant Index	Set
1	{KIND, SPEED, WHEELS, WINGSPAN}
2	{KIND, SPEED, WHEELS}
3	{KIND, SPEED, STEAM}
4	{KIND, SPEED, STAGES}

A comparison between the variant index of a record value and the bounds of an interval is enough to check that a given component is present in the value:

Component	Interval
KIND	--
SPEED	--
WHEELS	1 .. 2
WINGSPAN	1 .. 1
STEAM	3 .. 3
STAGES	4 .. 4

The implicit component VARIANT_INDEX must be large enough to store the number V of component lists that don't contain variant parts. The compiler treats this implicit component as having an anonymous integer type whose range is 1 .. V.

If R is the name of the record type, this implicit component can be denoted in a component clause by the implementation generated name R'VARIANT_INDEX.

- *ARRAY_DESCRIPTOR*

An implicit component of this kind is associated by the compiler with each record component whose subtype is an anonymous array subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind ARRAY_DESCRIPTOR is not described in this documentation. Nevertheless, if a programmer is interested in specifying the location of a component of this kind using a component clause, he can obtain the size of the component using the ASSEMBLY parameter in the COMPILE command.

The compiler treats an implicit component of the kind ARRAY_DESCRIPTOR as having an anonymous array type. If C is the name of the record component whose subtype is described by the array descriptor, then this implicit component can be denoted in a component clause by the implementation generated name C'ARRAY_DESCRIPTOR.

- *RECORD_DESCRIPTOR*

An implicit component of this kind is associated by the compiler with each record component whose subtype is an anonymous record subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind RECORD_DESCRIPTOR is not described in this documentation. Nevertheless, if a programmer is interested in specifying the location of a component of this kind using a component clause, he can obtain the size of the component using the ASSEMBLY parameter in the COMPILE command.

The compiler treats an implicit component of the kind RECORD_DESCRIPTOR as having an anonymous array type. if C is the name of the record component whose subtype is described by the record descriptor, then this implicit component can be denoted in a component clause by the implementation generated name C'RECORD_DESCRIPTOR.

Suppression of implicit components

The Alsys implementation provides the capability of suppressing the implicit components RECORD_SIZE and/or VARIANT_INDEX from a record type. This can be done using an implementation defined pragma called IMPROVE. The syntax of this pragma is as follows:

```
    pragma IMPROVE ( TIME | SPACE , [ON =>] simple_name );
```

The first argument specifies whether TIME or SPACE is the primary criterion for the choice of the representation of the record type that is denoted by the second argument.

If TIME is specified, the compiler inserts implicit components as described above. If on the other hand SPACE is specified, the compiler only inserts a VARIANT_INDEX or a RECORD_SIZE component if this component appears in a record representation clause that applies to the record type. A record representation clause can thus be used to keep one implicit component while suppressing the other.

A pragma IMPROVE that applies to a given record type can occur anywhere that a representation clause is allowed for this type.

Size of a record subtype

Unless a component clause specifies that a component of a record type has an offset or a size which cannot be expressed using storage units, the size of a record subtype is rounded up to the a whole number of storage units.

The size of a constrained record subtype is obtained by adding the sizes of its components and the sizes of its gaps (if any). This size is not computed at compile time

- when the record subtype has non-static constraints,
- when a component is an array or a record and its size is not computed at compile time.

The size of an unconstrained record subtype is obtained by adding the sizes of the components and the sizes of the gaps (if any) of its largest variant. If the size of a component or of a gap cannot be evaluated exactly at compile time an upper bound of this size is used by the compiler to compute the subtype size.

A size specification applied to a record type or first named subtype has no effect. The only size that can be specified using such a length clause is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of a record is as expected by the application.

Size of an object of a record subtype

An object of a constrained record subtype has the same size as its subtype.

An object of an unconstrained record subtype has the same size as its subtype if this size is less than or equal to 8 kb. If the size of the subtype is greater than this, the object has the size necessary to store its current value; storage space is allocated and released as the discriminants of the record change.

Alignment of a record subtype

When no record representation clause applies to its base type, a record subtype is even byte aligned if it contains a component whose subtype is even byte aligned. Otherwise the record subtype is byte aligned.

When a record representation clause that does not contain an alignment clause applies to its base type, a record subtype is even byte aligned if it contains a component whose subtype is even byte aligned and whose offset is a multiple of 16 bits. Otherwise the record subtype is byte aligned.

When a record representation clause that contains an alignment clause applies to its base type, a record subtype has an alignment that obeys the alignment clause An alignment clause can specify that a record type is byte aligned or even byte aligned.

Address of an object of a record subtype

Provided its alignment is not constrained by a representation clause, the address of an object of a record subtype is even when its subtype is even byte aligned.

CHAPTER 5

IMPLEMENTATION-DEPENDENT COMPONENTS

The following forms of implementation-generated names [13.4(8)] are used to denote implementation-dependent record components, as described in Section 4.8 in the paragraph on indirect and implicit components:

C'OFFSET
R'RECORD_SIZE
R'VARIANT_INDEX
R'ARRAY_DESCRIPTORs
R'RECORD_DESCRIPTORs

where C is the name of a record component and R the name of a record type.

CHAPTER 6

ADDRESS CLAUSES

An address clause can be used to specify the address of an object, a program unit or an entry.

6.1 Address Clauses for Objects

An address clause can be used to specify an address for an object as described in [13.5]. When such a clause applies to an object no storage is allocated for it in the program generated by the compiler. The program accesses the object by using the address specified in the clause.

An address clause is not allowed for task objects, for unconstrained records whose size is greater than 8 kb, or for a constant.

Note that the function SYSTEM.VALUE, defined in the package SYSTEM, is available to convert a STRING value into a value of type SYSTEM.ADDRESS, also, the IMPORT attribute is available to provide the address of an external symbol. (Refer to Chapter 3 and section 2.3)

6.2 Address Clauses for Program Units

Address clauses for program units are not implemented in the current version of the compiler.

6.3 Address Clauses for Entries

Address clauses for entries are not supported in the current version of the compiler.

CHAPTER 7

UNCHECKED CONVERSIONS

Unchecked type conversions are described in [13.10.2]. The following restrictions apply to their use.

Unconstrained arrays are not allowed as target types. Unconstrained record types without defaulted discriminants are not allowed as target types. Access types to unconstrained arrays are not allowed as target or source types. Note also that UNCHECKED_CONVERSION cannot be used for an access to an unconstrained string.

However, if the source and the target types are each scalar or access types, the sizes of the objects of the source and target types must be equal.

If a composite type is used either as source type or as target type this restriction on the size does not apply.

If the source and the target types are each of scalar or access type or if they are both of composite type, the effect of the function is to return the operand.

In other cases the effect of unchecked conversion can be considered as a copy:

- If an unchecked conversion is achieved of a scalar or access source type to a composite target type, the result of the function is a copy of the source operand. The result has the size of the source.

- If an unchecked conversion is achieved of a composite source type to a scalar or access target type, the result of the function is a copy of the source operand. The result has the size of the target.

CHAPTER 8

INPUT-OUTPUT CHARACTERISTICS

In this part of the Appendix the implementation-specific aspects of the input-output system are described.

8.1 Introduction

In Ada, input-output operations are considered to be performed on *objects* of a certain file type rather than being performed directly on external files. An external file is anything external to the program that can produce a value to be read or receive a value to be written. Values transferred for a given file must be all of one type.

Generally, in Ada documentation, the term *file* refers to an object of a certain file type, whereas a physical manifestation is known as an *external file*. An external file is characterized by

- its NAME, which is a string defining a legal path name under the current version of the operating system
- its FORM, which gives implementation-dependent information on file characteristics.

Both the NAME and the FORM appear explicitly as parameters of the Ada procedures CREATE and OPEN. Though a file is an object of a certain file type, ultimately the object has to correspond to an external file. Both CREATE and OPEN associate a NAME of an external file (of a certain FORM) with a program file object.

Ada input-output operations are provided by means of standard packages ([14]):

SEQUENTIAL_IO	A generic package for sequential files of a single element type.
DIRECT_IO	A generic package for direct (random) access files.

TEXT_IO	A generic package for human-readable files (text, ASCII). Note that the notion of standard input and output files is not appropriate for standalone applications. The exception USE_ERROR will be raised if a standalone applications attempts to use either TEXT_IO.STANDARD_INPUT or TEXT_IO.STANDARD_OUTPUT.
IO_EXCEPTIONS	A package which defines the exceptions needed by the above three packages.

The generic package LOW_LEVEL_IO is not implemented in this version.

The upper bound for index values in DIRECT_IO and for line, column and page numbers in TEXT_IO is given by

```
COUNT'LAST = 2**31 -1
```

The upper bound for field widths in TEXT_IO is given by

```
FIELD'LAST = 255
```

8.2 The Parameter FORM

The parameter FORM of both the procedures CREATE and OPEN in Ada specifies the characteristics of the external file involved.

The procedure CREATE establishes a new external file, of a given NAME and FORM, and associates it with a specified program file object. The external file is created (and the file object set) with a specified (or default) file mode. If the external file already exists, the file will be erased. The exception USE_ERROR is raised if the file mode is IN_FILE.

Example:

```
CREATE(F, OUT_FILE, NAME => "MY_FILE");
```

The procedure OPEN associates an existing external file, of a given NAME and FORM, with a specified program file object. The procedure also sets the current file mode. If there is an inadmissible change of mode, then the exception USE_ERROR is raised.

The parameter FORM is a string, formed from a list of attributes, with attributes separated by commas. The string is not case sensitive (so that, for example, *HERE* and *here* are treated alike). (FORM attributes are distinct from Ada attributes.) The attributes specify:

- File type
- File creator
- File structure
- Buffering
- Appending

The general form of each attribute is a keyword followed by => and then a qualifier. The *arrow and qualifier may sometimes be omitted.* The format for an attribute specifier is thus either of

KEYWORD

KEYWORD = > QUALIFIER

We will discuss each attribute in turn.

File Type

The keyword TYPE_ID may be used to define the four of byte type ID for a file. By default text files have the ID "TEXT", other files have the ID "????". The qualifier is required and is an optionally quoted string.

If the qualifier is not quoted and begins with a dollar sign, '$', it is interpreted as a 32 bit hexadecimal value.

Otherwise it is interpreted as a literal string. The string value, excluding quotes, must not be larger than four characters. If less than four characters it is left justified and blank filled.

File Creator

The keyword CREATOR_ID may be used to define the four of byte creator ID for a file. By default text files have the ID "   " (four spaces), other files have the ID "????". The qualifier is required and is an optionally quoted string.

If the qualifier is not quoted and begins with a dollar sign, '$', it is interpreted as a 32 bit hexadecimal value.

Otherwise it is interpreted as a literal string. The string value, excluding quotes, must not be larger than four characters. If less than four characters it is left justified and blank filled.

File Structure

(a) Text Files

There is no FORM attribute to define the structure of text files.

A text file consists of a sequence of bytes holding the ASCII codes of characters.

The representation of Ada terminators depends on the file's mode (IN or OUT) and whether it is associated with a terminal device or a mass storage file; the terminators are implicit in some cases, the characters present explicitly being as follows:

- Mass storage files and terminal device with mode OUT

end of line:	ASCII.CR
end of page:	ASCII.CR ASCII.FF

  The file length determines implicit page and file terminators at the end.

- Terminal device with mode IN

end of line:	ASCII.CR
end of page:	ASCII.FF
end of file:	Command-Enter

  The FF implies a line terminator; the end of file character implies both line and page terminators.

(b) Binary Files

Two FORM attributes, *RECORD_SIZE* and *RECORD_UNIT*, control the structure of binary files.

A binary file can be viewed as a sequence (sequential access) or a set (direct access) of consecutive RECORDS.

The structure of such a record is:

    [HEADER] OBJECT [UNUSED_PART]

and it is formed from up to three items:

- An OBJECT with exactly the same binary representation as the Ada object in the executable program, possibly including an object descriptor
- A HEADER consisting of two fields (each of 32 bits):
  - the length of the object in bytes (except for the length of unconstrained arrays which is in bits)
  - the length of the descriptor in bytes which is always set to 0
- An UNUSED_PART of variable size to permit full control of the record's size.

The HEADER is implemented only if the actual parameter of the instantiation of the input-output package is unconstrained.

The file structure attributes take the form:

RECORD_SIZE => size_in_bytes

RECORD_UNIT => size_in_bytes

Their meaning depends on the object's type (constrained or not) and the file access mode (sequential or direct access):

a) If the object's type is constrained:

- The attribute *RECORD_UNIT* is illegal
- If the attribute *RECORD_SIZE* is omitted, no UNUSED_PART will be implemented: the default *RECORD_SIZE* is the object's size
- If present, the attribute *RECORD_SIZE* must specify a record size greater than or equal to the object's size, otherwise the exception USE_ERROR will be raised

b) If the object's type is unconstrained and the file access mode is direct:

- The attribute *RECORD_UNIT* is illegal
- The attribute *RECORD_SIZE* has no default value, and if it is not specified, USE_ERROR will be raised
- An attempt to input or output an object larger than the given *RECORD_SIZE* will raise the exception DATA_ERROR

c) If the object's type is unconstrained and the file access mode is sequential:

- The attribute *RECORD_SIZE* is illegal
- The default value of the attribute *RECORD_UNIT* is 1 (byte)
- The record size will be the smallest multiple of the specified (or default) *RECORD_UNIT* that holds the object and its length. This is the only case where records of a file may have different sizes.

Buffering

The buffer size can be specified by the attribute

BUFFER_SIZE => size_in_bytes

A buffer size of 0 means no buffering.

The default value for buffer size depends on the type of the external file and on the file access mode, as follows:

- If the external file is a "regular" UNIX mass storage file, the default buffer size is the system's Input-Output block size (typically 1024 or 2048). For other types of UNIX files (directories, device files, named pipes), the default buffer size is 0 (no buffering).
- For a file used in direct access mode or the STANDARD_OUTPUT file, the default buffer size is in any case 0.

Appending

Only to be used with the procedure OPEN, the format of this attribute is simply

APPEND

and it means that any output will be placed at the end of the named external file.

In normal circumstances, when an external file is opened, an index is set which points to the beginning of the file. If the attribute *APPEND* is present for a sequential or for a text file, then data transfer will commence at the end of the file. For a direct access file, the value of the index is set to one more than the number of records in the external file.

This attribute is not applicable to terminal devices.

USE_ERROR is raised when in mode IN_FILE.

USE_ERROR is raised if the file size is not a multiple of RECORD_SIZE or RECORD_UNIT.

INDEX